高性能プログラミング（Ⅱ）上級編

黒田 久泰
東京大学情報基盤センター

1. はじめに

高い性能を出すプログラムを作成するには、システムの特徴をよく理解しておく必要があります。本稿では、HA8000 クラスタシステムで高い性能を出すプログラムを作成する方法について述べます。

2. クロックレベル時間計測

プログラムの高速化を行う上で、高い精度での時間計測を行う方法を知っておくと便利です。

ここでは、Intel（Pentium 以降）や AMD のプロセッサで利用できる RDTSC（ReaD Time Stamp Counter）命令を利用する方法を紹介します。この RDTSC 命令は 64bit のタイムスタンプカウンタの値（システム起動時からの CPU クロック数）を返します。C 言語で RDTSC 命令を使うには、インラインアセンブラを用います（そのため日立最適化コンパイラでは利用できません)。次のようにマクロ定義を利用すると使い易くなります。ここで、仮引数 X には符号なし 64bit 整数型の変数（unsigned long long で宣言した変数）を指定します。

```
#define RDTSC(X)¥
   asm volatile ("rdtsc; shlq $32,%%rdx; orq %%rdx,%%rax" : "=a" (X) :: "%rdx")
```

volatile はコンパイラの最適化を防ぐためのキーワードです。RDTSC 命令を実行すると 64bit レジスタである rdx と rax に値が入ってきますが、それぞれ下位の 32bit に値が格納されています。rdx の値を左に 32bit 論理シフトし、rax と論理和を取り、仮引数 X に代入するということを行っています。

IA-32 アーキテクチャの場合や、gcc で -m32 オプションを付けてコンパイルする場合には、アセンブラコードが少し変わってきますので、下記のように定義して下さい。

```
#define RDTSC(X) asm volatile ("rdtsc" : "=A" (X))
```

IA-32 アーキテクチャでは、32 ビットレジスタである edx と eax に値が格納されます。RDTSC 命令が返す値の有効なビット数は x86_64 モードと同じ 64 ビットとなりますので、仮引数 X に符号なし 64bit 整数型の変数を代入する点は同じです。

RDTSC 命令のマクロ定義を利用したプログラム例は次のとおりです。

```
   unsigned long long t1, t2;
   RDTSC(t1);
   // 時間計測したいコードをここに挿入する
   RDTSC(t2);
   printf("CPU Clock=%llu¥n", t2-t1);
```

HA8000 クラスタシステムでは、1 ノード上に 4 つのプロセッサが載っていますが、RDTSC 命令の返す値はほぼ等しくなるようになっています。

最近のプロセッサでは CPU の動作周波数を変更した場合でも、RDTSC の返すクロックの周期は一定に保たれるようになっています。Intel Pentium M では CPU の動作周波数を変更すると、RDTSC の返すクロックの周期も変化します。クロック値を秒に変換するような場合にはこれらの違いに注意して下さい。

サンプルプログラム

RDTSC 命令を使った例として、for ループに要した時間をクロック数で計測するプログラムを紹介します。for ループの各反復で「a=a+a」を 10 回行うもの、「a=a*a」を 10 回行うものを計測します。

```
#include <stdio.h>
#include <stdlib.h>
#define RDTSC(X) asm volatile ("rdtsc;shlq $32,%%rdx;orq %%rdx,%%rax":"=a"(X)::"%rdx")
#define N 10000

int main(int argc, char *argv[])
{
   unsigned long long i, t1, t2;
   double a = atof(argv[1]);

   RDTSC(t1);
   for(i = 0; i < N; i++){
     a=a+a; a=a+a; a=a+a; a=a+a; a=a+a;
     a=a+a; a=a+a; a=a+a; a=a+a; a=a+a;
   }
   RDTSC(t2);
   printf("a=%e¥n", a);
   printf("CPU Clock=%llu  /N=%.1f¥n", t2-t1, (double)(t2-t1)/N);

   RDTSC(t1);
   for(i = 0; i < N; i++){
     a=a*a; a=a*a; a=a*a; a=a*a; a=a*a;
     a=a*a; a=a*a; a=a*a; a=a*a; a=a*a;
   }
   RDTSC(t2);
   printf("a=%e¥n", a);
   printf("CPU Clock=%llu  /N=%.1f¥n", t2-t1, (double)(t2-t1)/N);
}
```

変数 a の初期値をプログラム内で決めると、最適化によりコンパイル時に計算してしまうことがあるため、実行時に引数として与えるようにしています。プログラムの実行は、「./a.out 0」のようにします。コンパイラやコンパイルオプションを変えて実行した結果は次のとおりです。右 2 列の数値はクロック数を表しています。

コンパイルの方法	「a=a+a」を 10 回	「a=a*a」を 10 回
gcc -O0	129	130
gcc -O1 (gcc -O と同じ)	50	51
gcc -O2 / gcc -O3	40	40
icc -O0	130	145
icc -O1	130	129
icc -O2 (icc -O と同じ) / icc -O3	40	40

よく利用される「gcc -O」では若干性能が劣ります。gcc を使う場合には、-O2 か -O3 を指定した方が良さそうです。Opteron においては、SSE2 命令の addsd と mulsd のレジスタ同士の演算のレイテンシーは 4 クロックですので、「a=a+a」と「a=a*a」を繰り返し実行する場合、1 回あたり 4 クロックかかります。10 回で 40 クロックというのは、理論上は最大性能が出ているということになります。なお、Intel 系のプロセッサでは、レイテンシーが異なりますので、上記とはまた違った結果になります。

3. プログラムを特定の CPU で実行させる方法

HA8000 クラスタシステムでは、各ノードには 4 プロセッサが搭載されており、その各プロセッサには 4CPU が搭載されていますので、合計 16CPU が搭載されています。通常、プロセスやスレッドがどの CPU で実行されるかは決まっていないため、実行する CPU は切り替わっていきます。ここでは、スレッドやプロセスを常にある特定の CPU で実行させる方法について紹介します。

プロセスを特定の CPU で実行させるには、システムコールの sched_setaffinity()を利用します。1 番目の引数にはプロセス ID またはスレッド ID を指定します。ここの値で 0 を指定すると、この関数を呼び出したプロセス（あるいはスレッド）が対象となりますので、通常は 0 を指定してください。2 番目の引数には CPU の集合のサイズ（バイト数）を指定しますが、通常は sizeof(cpu_mask)で構いません。3 番目の引数には、実行する CPU の集合を指定します。この CPU の集合を操作するためのマクロが用意されています。CPU_ZERO は CPU の集合をクリアします。CPU_SET は指定された CPU 番号(HA8000 クラスタシステムでは 0 から 15)を CPU 集合に加えます。

```
 1: #define _GNU_SOURCE
 2: #include <unistd.h>
 3: #include <sched.h>
 4:
 5: int main(void)
 6: {
 7:    int rv;
 8:    cpu_set_t cpu_mask;
 9:
10:    CPU_ZERO(&cpu_mask);
11:    CPU_SET(10, &cpu_mask);
12:    rv = sched_setaffinity((pid_t)0, sizeof(cpu_mask), &cpu_mask);
13:    // ここから先は 10 番の CPU 上で実行される
14: }
```

上記の例では、10 行目で CPU 集合をクリアし、11 行目で CPU 10 番を CPU 集合に加えています。この場合、常に CPU 10 番で実行されるようになります。CPU_SET で指定する CPU 番号を変えれば、実行する CPU を変更することができます。

これらの詳しい使い方についてはオンラインマニュアルをご覧ください。HA8000 クラスタシステムのログインノード上で「man sched_setaffinity」を実行すると詳細が表示されます。

なお、CPU 数はシステムによって異なりますので、移植性の高いプログラムを作成する場合には、CPU 数を調べた上で、どの CPU に割り当てるかを決める必要が出てきます。CPU 数を取得するには、下記のように get_nprocs() 関数を利用します。

```
1: #include <sys/sysinfo.h>
2:
3: int main(void)
4: {
5:   int n;
6:   n = get_nprocs();
7:   printf("The number of CPUs = %d¥n", n);
8: }
```

get_nprocs() 関数については、オンラインマニュアルにはありません。このようにオンラインマニュアルにない関数は「man undocumented」や「man 3 undocumented」を実行すると表示されます。

4. メモリを固定した場所に割り付ける方法

HA8000 クラスタシステムでは、メモリは 4 つに分割されて配置されています。その 4 つの分割されているメモリは 4 つのプロセッサと対応しています。固定的にメモリを割り付けるには、システムコールである mmap()と syscall()を利用します。

mmap()では、指定したサイズの領域を仮想メモリ空間に確保します。

【書式】

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

【説明】

addr には開始アドレスを指定しますが、通常は NULL を指定して OS に任せます。length には確保する領域のバイト数を指定します。prot には確保した領域のメモリ保護をどうするかを指定しますが、通常は PROT_READ|PROT_WRITE となります。flags には他のプロセスと共有するかどうかといったいろいろな属性を指定しますが、今回の目的では MAP_PRIVATE|MAP_ANONYMOUS とします。fd にはファイルディスクリプタ、offset にはファイルのオフセット値を指定しますが、flags で MAP_ANONYMOUS を指定している場合には、fd は-1、offset は 0 を指定します。mmap()の戻り値は確保した仮想メモリ空間のアドレスの開始位置となります。この開始位置は、ページサイズ（HA8000 クラスタシステムでは 4K バイト）の倍数となります。仮想メモリ空間への確保に失敗した場合は-1 を返します。

今回使う mmap()は、標準 C ライブラリの malloc() で代用することも可能です。しかし、malloc()で割り当てられるメモリ空間の開始アドレスはページサイズ境界（4K バイト単位）と一致していないため、このままでは物理メモリ空間に割り当てることができません。malloc()を使った場合でも、内部的には mmap()を利用しているので、今回は最初から mmap()を使う方法を説明しています。

mmap()で仮想メモリ空間に割り当てた後、物理メモリにバインドします。そのためには syscall()を利用します。

【書式】

```
#include <sys/syscall.h>
#include <unistd.h>
int syscall(int number, ...);
```

number にはどのシステムコールを利用するかを指定します。その後の引数の個数は、number 毎に異なります。ここでは、number に SYS_mbind を指定します。

【書式】

```
int syscall(SYS_mbind, void *start, unsigned long len, int policy, unsigned long *nodemask,
            unsigned long maxnode, unsigned flags);
```

【説明】

start から始まる長さ len バイトの範囲のメモリに NUMA メモリポリシー（policy）を設定します。policy には、通常、MPOL_BIND（メモリ割り当ては nodemask に指定されたノードに限定するという意味）を指定します。nodemask にはノードのビットマスクを指定し、maxnode には最大のノードを指定します。flags には各種属性を指定します。なお、ここでのノードは、NUMA で一般的に使われているノードのことで、4 つに分割されたプロセッサとメモリの対のことを指します。HA8000 クラスタシステムの各計算ノードは、NUMA の定義においては 4 ノードで構成されています。nodemask は、0

番ノードは 0x1、1 番ノードは 0x2、2 番ノードは 0x4、3 番ノードは 0x8 を指定します。複数のノードを指定する場合には、それらの値を合計した値を指定します。

サイズが 100 万バイトのメモリ領域を NUMA でいうところの 0 番ノードに割り当てるプログラムは次のようになります。12 行目で仮想メモリ空間に領域を確保します。13 行目でノードのビットマスクを作成しますが、ここでは 0 番ノードに限定させるため、mem_mask に 0x1 を代入しています。14 行目で実際に仮想メモリ空間のアドレス a から SIZE バイトの領域を物理メモリ空間にバインドします。高度な使い方として、SIZE バイトの領域を分割して syscall()を複数回実行するようなことも可能です。最後に 16 行目では仮想メモリ空間に確保した領域を解放しています。

```
 1: #include <sys/mman.h>
 2: #include <sys/syscall.h>
 3: #include <unistd.h>
 4: #include <numaif.h>
 5: #define SIZE 1000000
 6:
 7: int main(void)
 8: {
 9:     char *a;
10:     unsigned long mem_mask;
11:
12:     a = mmap(NULL, SIZE, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
13:     mem_mask = 0x1;
14:     syscall(SYS_mbind, a, SIZE, MPOL_BIND, &mem_mask, 0x8, MPOL_MF_STRICT);
15:
16:     munmap(a, SIZE);
17: }
```

なお、メモリ領域をノード 0 番に固定した場合、そのメモリに頻繁にアクセスするプロセスについても、実行する CPU をノード 0 番にあるプロセッサ（CPU 0 番から 3 番）に固定しないと効果がありません。NUMA でいうところのノードと CPU 番号の対応は下記のようになります。

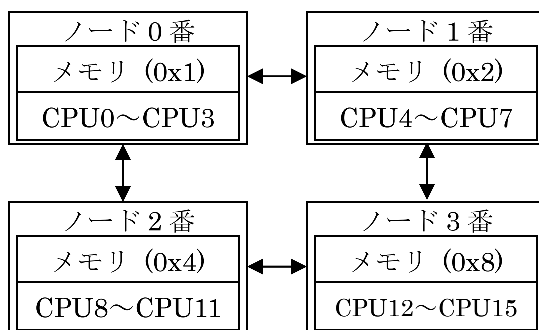

図 1　NUMA ノードと CPU 番号の対応

サンプルプログラム

約 1GB のメモリ領域を指定したノードに確保し、指定した CPU からそのメモリ領域に値（0x77）を書き込むプログラムを紹介します。実行する CPU とメモリを割り当てたノードの位置関係で速度が大きく変わることが確認できます。

【プログラムの実行方法】

$ ./a.out　実行する CPU 番号（0～15）　メモリを確保するノード番号（0～3）

```
#define _GNU_SOURCE
#include <sys/syscall.h>
#include <sys/mman.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sched.h>
#include <numaif.h>

#define N 1000000000
#define RDTSC(X) asm volatile ("rdtsc;shlq $32,%%rdx;orq %%rdx,%%rax":"=a"(X)::"%rdx")

int main(int argc, char *argv[])
{
  char *a;
  int i, cpu, memorynode;
  unsigned long mem_mask;
  unsigned long long t1, t2;
  cpu_set_t cpu_mask;
  if (argc != 3) return 0;
  cpu = atoi(argv[1]);   memorynode = atoi(argv[2]);

  CPU_ZERO(&cpu_mask);   CPU_SET(cpu, &cpu_mask);
  sched_setaffinity((pid_t)0, sizeof(cpu_mask), &cpu_mask);

  a = mmap(NULL,N, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
  mem_mask = 0x1 << memorynode;
  syscall(SYS_mbind, a, N, MPOL_BIND, &mem_mask, 0x8, MPOL_MF_STRICT);

  for(i = 0; i < 2; i++){
    RDTSC(t1);   memset(a, 0x77, N);   RDTSC(t2);
    printf("cpu=%2d  MemoryNode=%d  Memory Bandwidth=%5.0f MB/s¥n",
           cpu, memorynode, (double)N/1024/1024/((double)(t2-t1)/2300096000.0));
  }
  munmap(a, N);
  return 0;
}
```

実行結果は次にようになります。ノード1番にある CPU 4 からの書き込みが 3258MB/s と一番速いことがわかります。一方、ノード2番にある CPU 8 からの書き込みは 2056MB/s しかありません。このようにノード1番とノード2番は、NUMA 上は距離が離れていることに注意してください。プロセスを実行する CPU がノード1番から順に離れていくと 1：1.28〜1.30：1.58 という性能比（数値が大きいとメモリアクセスが遅い）となり、その差が大きいことがわかります。

```
$ ./a.out 0 1
cpu= 0  MemoryNode=1  Memory Bandwidth=  648 MB/s
cpu= 0  MemoryNode=1  Memory Bandwidth= 2508 MB/s
$ ./a.out 4 1
cpu= 4  MemoryNode=1  Memory Bandwidth=  966 MB/s
cpu= 4  MemoryNode=1  Memory Bandwidth= 3258 MB/s
$ ./a.out 8 1
cpu= 8  MemoryNode=1  Memory Bandwidth=  583 MB/s
cpu= 8  MemoryNode=1  Memory Bandwidth= 2056 MB/s
$ ./a.out 12 1
cpu=12  MemoryNode=1  Memory Bandwidth=  663 MB/s
cpu=12  MemoryNode=1  Memory Bandwidth= 2550 MB/s
```

5. SSE 命令の使い方

5.1　組み込み関数

HA8000 クラスタシステムで採用している Opteron プロセッサはストリーミング SIMD 拡張命令の SSE3 を搭載しています。SSE は Intel の Pentium や Pentium II に採用された MMX（MultiMedia eXtension）の後継にあたり、Pentium III 以降の CPU に搭載されています。SSE2 は Pentium 4 及び AMD の Athlon64 から搭載されていますが、SSE2 からは 2 個の倍精度浮動小数点演算をまとめて処理できるようになりました。SSE3 は 2004 年 2 月に登場した Prescott コアの Pentium 4 以降の CPU に搭載されており、AMD のほとんどの CPU でも利用できます。SSE3 ではメモリアクセス及び複素数計算（1 つの命令で加算と減算が同時に行える）の高速化が図られています。

SSE2 / SSE3 では、32bit モード時は 8 個の XMM レジスタ、64bit モード時は 16 個の XMM レジスタが利用できます。この XMM レジスタは 128bit 長であり、1 つの XMM レジスタに倍精度浮動小数点値を 2 つ格納して同時に処理することができます。

SSE 命令を直接 C 言語から利用するには、コンパイラで用意されている組み込み関数（イントリンシック（intrinsic） 関数とも言う）を利用します。この組み込み関数は gcc、Intel C / C++コンパイラ、Microsoft Visual C++コンパイラで共通仕様となっています。組み込み関数を利用するには、下記のように SSE3 用の組み込み関数用ヘッダファイル pmmintrin.h をインクルードします。

```
#include <pmmintrin.h>
```

HA8000 クラスタシステム上の gcc でコンパイルする場合、SSE3 の機能を利用するには -msse3 を指定してコンパイルしてください。デフォルト設定ではこのオプションは付加されておりません。

```
gcc -O3 -msse3 ソースファイル名
```

SSE2 の機能だけを利用する場合

SSE3 の機能を利用しない場合は、SSE2 組み込み関数用ヘッダファイル emmintrin.h をインクルードします。SSE3 を搭載していない CPU 上で実行することが可能となります。

```
#include <emmintrin.h>
```

HA8000 クラスタシステムの gcc コンパイラでは、デフォルト設定で -msse2 が設定されていますので、特に何も指定する必要はありません。

ここからは、倍精度浮動小数点演算を利用する場合について説明します。128bit 長の変数（XMM レジスタに対応）を宣言するために、__m128d 型（パックド倍精度浮動小数点型）というものを利用します。この型で宣言した変数 1 つに倍精度浮動小数点値を 2 つ格納することができます。

__m128d 型（パックド倍精度浮動小数点型）

C 言語では下記のように記述します。

```
__m128d x1;
```

このように宣言すると、変数 x1 には 2 つの倍精度浮動小数点値を格納できます。変数宣言時には「__m128d x1 = { 0.0, 0.0 };」のように初期値を代入することができます。

SSE2 の組み込み関数で倍精度浮動小数点演算に関するものの一部を表 1 に示します。これが全てというわけではありませんのでご注意ください。

表１　倍精度浮動小数点演算用の組み込み関数（SSE2 用）

組み込み関数	説明	対応する SSE2 命令
X = _mm_load_pd(p)	2 つの倍精度浮動小数点値をセットする X0 = p[0]　X1 = p[1]　※p は 16 バイト境界であること	movapd
X = _mm_load1_pd(p)	2 つの倍精度浮動小数点値に同じ値をセットする X0 = *p　X1 = *p　※p に制約なし	movsd+unpcklpd SSE3 は movddup
X = _mm_loadr_pd(p)	2 つの倍精度浮動小数点値を逆順でセットする X0 = p[1]　X1 = p[0]　※p は 16 バイト境界であること	movapd+shufpd
X = _mm_loadu_pd(p)	2 つの倍精度浮動小数点値をセットする X0 = p[0]　X1 = p[1]　※p に制約なし	movupd
X = _mm_load_sd(p)	下位に値をセットし、上位は 0.0 にする X0 = *p　X1 = 0.0　※p に制約なし	movsd
X=_mm_loadh_pd(Y,p)	上位に値をセットする。下位は Y の下位を引き継ぐ X0 = Y0　X1 = *p　※p に制約なし	movhpd
X=_mm_loadl_pd(Y, p)	下位に値をセットする。上位は Y の上位を引き継ぐ X0 = *p　X1 = Y1　※p に制約なし	movlpd
X = _mm_add_pd(Y, Z)	X0 = Y0 + Z0　X1 = Y1 + Z1	addpd
X = _mm_sub_pd(Y, Z)	X0 = Y0 − Z0　X1 = Y1 − Z1	subpd
X = _mm_mul_pd(Y, Z)	X0 = Y0 * Z0　X1 = Y1 * Z1	mulpd
X = _mm_div_pd(Y, Z)	X0 = Y0 / Z0　X1 = Y1 / Z1	divpd
_mm_store_sd(p, X)	下位の倍精度浮動小数点値をストアします。 *p = X0　※p に制約なし	movsd
_mm_store_pd(p, X)	2 つの倍精度浮動小数点値をストアします。 p[0] = X0　p[1] = X1　※p は 16 バイト境界であること	movapd
_mm_storeu_pd(p, X)	2 つの倍精度浮動小数点値をストアします。 p[0] = X0　p[1] = X1　※p に制約なし	movupd
X = _mm_setzero_pd()	2 つの倍精度浮動小数点値を 0.0 にする X0 = 0.0　X1 = 0.0	xorpd
X = _mm_set_sd(s)	下位の倍精度浮動小数点値を s、上位は 0.0 にする X0 = s　X1 = 0.0	movsd
X = _mm_set1_pd(s)	2 つの倍精度浮動小数点値を s にする X0 = s　X1 = s	movapd
X = _mm_set_pd(s, t)	上位の倍精度浮動小数点値を s、下位は t にする X0 = t　X1 = s	movapd
X = _mm_setr_sd(s, t)	上位の倍精度浮動小数点値を t、下位は s にする X0 = s　X1 = t	movapd
X=_mm_move_sd(Y, Z)	Y の上位を X の上位に、Z の下位を X の下位にする X0 = Z0　X1 = Y1	movsd

※X, Y, Z は__m128d 型、s, t は double 型、p は double のポインタ型を表す
※X の上位の倍精度浮動小数点値を X1、下位の倍精度浮動小数点値を X0 と表記している

SSE3 の組み込み関数で倍精度浮動小数点演算に関するものは表 2 のとおりです。

_mm_addsub_pd() が加わったことで、複素数同士の積の演算に関して効率が良くなりました。また、_mm_loaddup_pd() は表 1 の_mm_load1_pd()と同じ機能ですが、movddup 命令 1 つで済むため高速化が実現されています。

表 2　倍精度浮動小数点演算用の組み込み関数（SSE3 用）

組み込み関数	説明	対応する SSE3 命令
X = _mm_addsub_pd(Y, Z)	上位に対して足し算、下位に対して引き算を行う X0 = Y0 - Z0　X1 = Y1 + Z1	addsubpd
X = _mm_hadd_pd(Y, Z)	xmm レジスタ内の 2 つの倍精度浮動小数点を足し算する X0 = Y0 + Y1　X1 = Z0 + Z1	haddpd
X = _mm_hsub_pd(Y, Z)	xmm レジスタ内の 2 つの倍精度浮動小数点を引き算する X0 = Y0 - Y1　X1 = Z0 - Z1	hsubpd
X = _mm_loaddup_pd(p)	2 つの倍精度浮動小数点値をセットする X0 = *p　X1 = *p　※p に制約なし	movddup
X = _mm_movedup_pd(Y)	Y の下位の倍精度浮動小数点値を複製してセットする X0 = Y0　X1 = Y0	movddup

5.2　演算性能

HA8000 クラスタシステムのプロセッサ AMD Quad-Core Opteron 8356（2.3GHz）は、1CPU あたり倍精度実数で 9.2GFlops の演算性能を持っています。これは、1 クロックに 4 つの浮動小数点演算が行えるということを意味しています。しかし、1 クロックに掛け算だけを 4 回、あるいは、足し算だけを 4 回行うといったことはできません。1 クロックに掛け算 2 回と足し算または引き算を 2 回実行できます。SSE3 命令を備えているため、掛け算 2 回、足し算 1 回、引き算 1 回も可能です。つまり、高い性能を出すためには、掛け算と加減算が同一回数である必要があります。

また、メインメモリへのアクセスが必要となる計算においては高い性能は達成できません。メモリアクセス性能がボトルネックとなるためです。そのため、高い性能を達成するためには、L1 キャッシュメモリ（命令キャッシュ、データキャッシュとも 64K バイトで分離されている）に載っているデータに対しての処理でなくてはなりません。

最後に、レイテンシーという概念も重要となります。倍精度実数の加算、減算、乗算の SSE2 / SSE3 のアセンブラ命令は addsd、addpd、subsd、subpd、mulsd、mulpd、addsubpd です。—sd は XMM レジスタの下位 64bit に格納されている倍精度実数だけを演算対象とします。—pd は XMM レジスタに格納されている 2 つの倍精度実数の両方を演算対象とします。これらの命令には、2 つの XMM レジスタ間で実行する場合に 4 クロックのレイテンシーがあります。具体的には、演算結果を格納する XMM レジスタに対し、その結果を参照できるようになるまで 4 クロック待たされるということを意味しています。そのため、この 4 クロックのレイテンシーを隠蔽するようなプログラムが必要になります。なお、同じ Opteron でもシングルコア時代の AMD K8 マイクロアーキテクチャに基づくものについては、2 つの倍精度実数演算をまとめて行う addpd、subpd、mulpd、addsubpd のレイテンシーは 5 クロックです。また、最近の Intel のプロセッサでは、addsd、addpd、subsd、subpd、addsubpd のレイテンシーは 3 クロック、mulsd、mulpd のレイテンシーは 5 クロックのように加減算と掛け算でレイテンシーが大きく異なっています。このようにレイテンシーはプロセッサの種類によって異なっていますので、高い性能を出すプログラムを作成する際には注意が必要です。

5.3　配列要素の総和

倍精度実数の配列要素の総和を求める次のプログラムについて考えます。また、本節以降、予め配列の全要素が L1 キャッシュ（64K バイト）に載っていることを前提とします。

```
 1: double dsum(double x[], int n)
 2: {
 3:    int i;
 4:    double s = 0.0;
 5:    for(i = 0; i < n; i++){
 6:       s += x[i];
 7:    }
 8:    return s;
 9: }
```

とても単純なプログラムなので、何もしなくても、加算だけを考慮した場合のピーク性能 4.6GFlops を達成できるように思われるかもしれません。しかし、「gcc -O3 -msse3」でコンパイルすると性能は 0.575GFlops（4.6GFlops の 8 分の 1）しか出ていないことがわかります。これは、1 回の反復で 1 回の倍精度実数演算しか行っていないこと、及び、1 回の倍精度実数同士の加算（addsd 命令）に 4 クロックもかかっていることが原因です。

SSE2 / SSE3 命令の組み込み関数を用いて記述すると次のプログラムのようになります。なお、プログラムを簡単にするため配列のサイズ n が 2 の倍数であると仮定しています。

```
 1: #include <pmmintrin.h>
 2:
 3: static double s[2] __attribute__ ((aligned(16)));
 4:
 5: double dsum_sse(double x[], int n)
 6: {
 7:    int i;
 8:    __m128d x0 = { 0.0,  0.0 };
 9:
10:    for(i = 0; i < n; i += 2){
11:       x0 = _mm_add_pd(x0, _mm_load_pd(&x[i]));
12:    }
13:    _mm_store_pd(s, x0);
14:    return s[0] + s[1];
15: }
```

1 行目では、SSE3 用の組み込み関数用ヘッダファイル pmmintrin.h をインクルードしています。ここでは、SSE3 固有の命令を利用していないので、SSE2 用の組み込み関数用ヘッダファイルである emmintrin.h をインクルードしても構いません。3 行目では、配列 s[2]を 16 バイト境界になるように宣言しています。8 行目でパックド倍精度浮動小数点型の変数 x0 を宣言し、初期値として 2 つの値にどちらも倍精度実数 0.0 を代入しています。11 行目で配列の要素を 2 つ読み込み、x0 の上位と下位にそれぞれ加えています。13 行目で、一旦 x0 の 2 つの倍精度実数の値を s[0]と s[1]に格納しています。最後に 14 行目で s[0] + s[1] の値を返しています。

上記のプログラムを「gcc -O3 -msse3」でコンパイルすると、1.15GFlops（4.6GFlops の 4 分の 1）になります。1 回で 2 つの倍精度実数の加算を行うようになったため、最初のプログラムと比べて性能が 2 倍になります。しかし、性能は満足できるものではありません。この理由は、XMM レジスタを 1 つしか用いておらず、レイテンシーを隠蔽できていないためです。

XMM レジスタを 4 つ使うように変更すると、理論的には最高性能となります。それには、プログラムを次のように書き換えます。なお、配列のサイズ n が 8 の倍数であると仮定しています。

```
 1: #include <pmmintrin.h>
 2:
 3: static double s[2] __attribute__ ((aligned(16)));
 4:
 5: double dsum_sse(double x[], int n)
 6:{
 7:    int i;
 8:    __m128d x0, x1, x2, x3;
 9:
10:    x0 = _mm_load_pd(x);
11:    x1 = _mm_load_pd(&x[2]);
12:    x2 = _mm_load_pd(&x[4]);
13:    x3 = _mm_load_pd(&x[6]);
14:
15:    for(i = 8; i < n; i += 8){
16:       x0 = _mm_add_pd(x0, _mm_load_pd(&x[i]));
17:       x1 = _mm_add_pd(x1, _mm_load_pd(&x[i+2]));
18:       x2 = _mm_add_pd(x2, _mm_load_pd(&x[i+4]));
19:       x3 = _mm_add_pd(x3, _mm_load_pd(&x[i+6]));
20:    }
21:
22:   x0 = _mm_add_pd(x0, x1);
23:   x2 = _mm_add_pd(x2, x3);
24:   x0 = _mm_add_pd(x0, x2);
25:   _mm_store_pd(s, x0);
26:   return s[0] + s[1];
27: }
```

「gcc -03 -msse3」でコンパイルすると、4.05GFlops（4.6GFlops の 88%）になります。XMM レジスタを 4 つ使った実装が一番高速になるのは HA8000 クラスタシステムに搭載の Opteron 8356 が AMD K10 マイクロアーキテクチャに基づくためです。AMD K8 マイクロアーキテクチャの場合には、XMM レジスタを 5 つ使うのが最適になります。一方、Intel のプロセッサの場合には、addpd 命令のレイテンシーが 3 クロックと小さいため、XMM レジスタは 3 つで十分ということになります。

5.4　二乗和の計算

ベクトルの 2-ノルムの計算などではこの二乗和を利用します。二乗和の計算では、加算と乗算の演算回数が等しいので、理論ピーク性能は 9.2GFlops ということになります。

```
 1: double dsumsq(double x[], int n)
 2: {
 3:    int i;
 4:    double s = 0.0;
 5:    for(i = 0; i < n; i++){
 6:       s += x[i] * x[i];
 7:    }
 8:    return s;
 9: }
```

「gcc -03 -msse3」でコンパイルすると、1.15GFlops（9.2GFlops の 8 分の 1）です。普通に書いたのではこのように性能が出ません。1 回の for ループに 4 クロックかかっているためです。

SSE2 / SSE3 命令の組み込み関数を用いて記述するとプログラムは次のようになります。

```
 1: #include <pmmintrin.h>
 2: static double s[2] __attribute__ ((aligned(16)));
 3: double dsumsq_sse(double x[], int n)
 4: {
 5:    int i;
 6:    __m128d x0;
 7:    __m128d x1 = {0.0, 0.0};
 8:
 9:    for(i = 0; i < n; i += 2){
10:       x0 = _mm_load_pd(&x[i]);
11:       x0 = _mm_mul_pd(x0, x0);
12:       x1 = _mm_add_pd(x1, x0);
13:    }
14:    _mm_store_pd(s, x1);
15:    return s[0] + s[1];
16: }
```

「gcc -O3 -msse3」でコンパイルすると、2.3GFlops（9.2GFlops の 4 分の 1）になります。これも、配列要素の総和で説明したのと同様にレイテンシーが隠蔽できていないためです。XMM レジスタを 8 つ利用したプログラムは次のようになります。

```
 1: double dsumsq_sse(double x[], int n)
 2: {
 3:    __m128d x0, x1, x2, x3;
 4:    __m128d x4 = {0.0, 0.0}, x5 = {0.0, 0.0};
 5:    __m128d x6 = {0.0, 0.0}, x7 = {0.0, 0.0};
 6:    for(i = 0; i < n; i += 8){
 7:       x0 = _mm_load_pd(&x[i]);
 8:       x1 = _mm_load_pd(&x[i+2]);
 9:       x2 = _mm_load_pd(&x[i+4]);
10:       x3 = _mm_load_pd(&x[i+6]);
11:       x0 = _mm_mul_pd(x0, x0);
12:       x1 = _mm_mul_pd(x1, x1);
13:       x2 = _mm_mul_pd(x2, x2);
14:       x3 = _mm_mul_pd(x3, x3);
15:       x4 = _mm_add_pd(x4, x0);
16:       x5 = _mm_add_pd(x5, x1);
17:       x6 = _mm_add_pd(x6, x2);
18:       x7 = _mm_add_pd(x7, x3);
19:    }
20:    x4 = _mm_add_pd(x4, x5);
21:    x6 = _mm_add_pd(x6, x7);
22:    x4 = _mm_add_pd(x4, x6);
23:    _mm_store_pd(s, x4);
24:    return s[0] + s[1];
25: }
```

レイテンシーが隠蔽されるようになったため、5.75GFlops (9.2GFlops の 4 分の 2.5) になります。最初のプログラムと比べると 5 倍速くなっています。

理論ピーク性能の 9.2GFlops までにはまだかなりの開きがありますが、L1 キャッシュメモリからの読み込みと演算部分がうまく隠蔽できていないことが原因で、さらに高い性能を得るためには直接アセンブラで記述する必要があります。

5.5 ベクトルの内積

ベクトルの内積の場合も加算と乗算の演算回数が等しいので、理論ピーク性能は 9.2GFlops になります。ただし、配列要素の二乗和と計算回数は同一ですが、メモリからの読み込みは 2 つの配列を対象とするため 2 倍必要になります。そのため、配列要素の二乗和と比べると、高い性能を出すのは難しくなります。特に L1 や L2 キャッシュメモリにデータが収まっていない場合には、ベクトルの内積の計算は、配列要素の二乗和に比べて 2 倍近く遅くなってしまいます。

```
 1: double ddot(double x[], double y[], int n)
 2: {
 3:    int i;
 4:    double s = 0.0;
 5:    for(i = 0; i < n; i++){
 6:       s += x[i] * y[i];
 7:    }
 8:    return s;
 9: }
```

「gcc -03 -msse3」でコンパイルすると、1.15GFlops（9.2GFlops の 8 分の 1）です。配列要素の二乗和と同じ性能となります。

SSE2 / SSE3 の組み込み関数を用いたプログラムは次のようになります。ここでは、1 回の for ループで 8 つの要素を計算しています。

```
 1: double ddot_sse(double x[],double y[], int n)
 2: {
 3:    int i;
 4:    __m128d x0, x1, x2, x3;
 5:    __m128d x4={0.0, 0.0}, x5={0.0, 0.0}, x6={0.0, 0.0}, x7={0.0, 0.0};
 6:
 7:    for(i = 0; i < n; i += 8){
 8:       x0 = _mm_mul_pd(_mm_load_pd(&x[i]), _mm_load_pd(&y[i]));
 9:       x1 = _mm_mul_pd(_mm_load_pd(&x[i+2]), _mm_load_pd(&y[i+2]));
10:       x2 = _mm_mul_pd(_mm_load_pd(&x[i+4]), _mm_load_pd(&y[i+4]));
11:       x3 = _mm_mul_pd(_mm_load_pd(&x[i+6]), _mm_load_pd(&y[i+6]));
12:       x4 = _mm_add_pd(x4, x0);
13:       x5 = _mm_add_pd(x5, x1);
14:       x6 = _mm_add_pd(x6, x2);
15:       x7 = _mm_add_pd(x7, x3);
16:    }
17:    x4 = _mm_add_pd(x4, x5);
18:    x6 = _mm_add_pd(x6, x7);
19:    x4 = _mm_add_pd(x4, x6);
20:    _mm_store_pd(s, x4);
21:    return s[0] + s[1];
22: }
```

「gcc -03 -msse3」でコンパイルすると、4.6GFlops（9.2GFlops の 2 分の 1）になります。最初のプログラムと比べると 4 倍速くなっています。しかし、配列要素の二乗和と比べると、1.25 倍の時間がかかっています。ベクトルの内積の場合も、L1 キャッシュメモリからの読み込みと演算部分のレイテンシーをうまく隠蔽するとさらに性能を向上させることができますが、プログラムはかなり複雑になります。

5.6　ベクトルとスカラーの積和（daxpy）

ベクトルとスカラーの積和計算 y = y + a＊x は数値計算の中でも比較的よく出てきます。BLAS レベル 1 のルーチン名が daxpy() となっているため、よく daxpy と記述されます。プログラムは下記のようになります。

```
 1: void daxpy(double y[], double a, double x[], int n)
 2: {
 3:    int i;
 4:    double s = 0.0;
 5:    for(i = 0; i < n; i++){
 6:       y[i] += a * x[i];
 7:    }
 8: }
```

「gcc -03 -msse3」でコンパイルすると、2.97GFlops（9.2GFlops の 32%）です。これまでのプログラムと比較すると、すでに高い性能が出ています。これは、ループ間に全く依存関係がないためです。つまり、1 個の XMM レジスタを使い回した場合でも、ループ同士がオーバーラップできるため、1 回の for ループに 4 クロックもかからなくなります。

SSE2 / SSE3 の組み込み関数を用いたプログラムは次のようになります。今回は、ループ間に依存関係がないので、XMM レジスタをそれほど多く使う必要はありません。

```
 1: void daxpy_sse(double a, double x[],double y[],int n)
 2: {
 3:    int i;
 4:     __m128d x0, x1, x2;
 5:
 6:    x0 = _mm_set1_pd(a);
 7:
 8:    for(i = 0; i < n; i += 2){
 9:       x1 = _mm_load_pd(&x[i]);
10:       x2 = _mm_load_pd(&y[i]);
11:       x1 = _mm_mul_pd(x1, x0);
12:       x2 = _mm_add_pd(x2, x1);
13:       _mm_store_pd(&y[i], x2);
14:    }
15: }
```

「gcc -03 -msse3」でコンパイルすると、5.29GFlops（ピーク性能の 58%）です。これについても、L1 キャッシュメモリからの読み込みと演算部分のレイテンシーをうまく隠蔽すると性能を向上させることができます。

L1 キャッシュメモリにデータが載っている場合には、演算にかかるレイテンシーをいかに隠蔽するかが重要となりますが、L2 キャッシュメモリにデータが載っている場合には、そこから XMM レジスタにデータが入ってくるまでのレイテンシーを隠蔽することも重要になってきます。

6. おわりに

HA8000 クラスタシステムは NUMA 構成になっているため、1 つの計算ノードの中にさらに 4 つのノードがあるということを意識したプログラミングが必要です。また、SSE2 / SSE3 命令を効果的に使うことも高速化には必要です。そのためには、基本演算については十分に最適化された数値計算ライブラリを用いるのが早道です。また、プログラム中で多くの時間を費やすのは特定の部分に集中していることも多いので、そういった部分を重点的に高速化することも効果的です。